Written Exercise Set 1 Solutions

Problem 1

1. b. Formula (b) is exactly the same as 1., as can be verified by computing it for all four combinations of a and b, i.e. 00, 01, 10, 11.
2. j. Negation of a number is the same as 2s complement.
3. g. Multiplication by a power of 2 is the same as shifting to the left by the exponent, which is 4 in this case, because 16 is 2^4.
4. a. TMIN + TMAX has value -1. Thus, -(TMIN+TMAX) is 1, and aˆ-(TMIN+TMAX) is aˆ1.
5. e. TMAX (zero followed by all ones) is the same as ~TMIN (one followed by all zeros).
6. c. This is also known as a ternary statement, which is shorthand for an if statement. When (a>=0), then the result in 1. This corresponds to the first part of the answer (a>>31), which has the same behavior. When (a<0), the answer is -1, which corresponds to the second term -((~a)>>31).

Descriptions of the unused right-column formulas:

"d". is a pure bitwise operation (i.e., it does the same thing to each bit position), and this combination of OR, AND, and NOT is equivalent to XOR: it starts with the inclusive OR `a | b`, but then it also requires either `a` or `b` to be false, which makes the case when both a and b are true give false.

"f". `1 << 31` is TMIN. Arithmetic right shift gives the value whose top two bits are 1 and the rest 0, `0xc0000000`.

"h". `1 << 30` is a value in which only the second-highest bit is set, `0x40000000`.

"i". It is the expression for division by 8 that rounds towards zero (floor rounding for positive inputs, ceiling rounding for negative inputs), which is the same as the C behavior of `a / 8`. Adding 7 when `a` is negative doesn't change the result when `a` is a multiple of 8, but it causes all other values to be rounded up towards 0 instead of down towards negative infinity.

"k". This expression firstly arithmetic shifts a by 31 times to the right. The result is all ones, when a is negative (because this is an arithmetic shift), and all zeros otherwise. Next, it shifts a to the left by 1, creating a zero at the LSB, whereas the remaining bits are either all ones or all zeros. Thus, this expression has value zero for non-negative numbers, and -2 for all negative numbers.

"l". This expression shifts a left 31 times, thus the MSB is set to 0 if a is even, and it is set to 1 when a is odd. Then, the expression is negated. Therefore, the result is -1 when a is even and TMAX when a is odd.

You will also see that most of these expressions are different when you plug in any random values for `a` and `b`. For instance if `a` is `0x0x80ff00fe` and `b` is `0xffff0000`, the various expressions give:

```
0x0ff00fe0 3: a * 16
0x0ff00fe0 g: a << 4
0x40000000 h: 1 << (31 - 1)
0x7f0000fe d: (a | b) & (~a | ~b)
0x7f00ff02 2: -a
0x7f00ff02 j: ~a + 1
0x7fffffff 5: TMAX
0x7fffffff e: ~TMIN
0x80ff00ff 4: a ^ 1
0x80ff00ff a: a ^ -(TMIN + TMAX)
0xc0000000 f: (1 << 31) >> 1
0xf01fe020 i: ((a < 0) ? (a + 7) : a) >> 3
0xffff00fe 1: a | b
```

```
0xffff00fe b: ~(~a & ~b )
0xfffffffe k: (a >> 31) << 1
0xffffffff 6: (a >= 0) ? 1 : -1
0xffffffff c: (a >> 31) - ((~a) >> 31)
0xffffffff l: ~(a << 31)
```

Problem 2

The table and functions for USatTimesTwo and TSatTimesTwo are shown below.

```
/*        times 2     USatTimesTwo    TSatTimesTwo */

/* 0000    0000            0000          0000 */
/* 0001    0010            0010          0010 */
/* 0010    0100            0100          0100 */
/* 0011    0110            0110          0110 */
/* 0100    1000            1000          0111 */
/* 0101    1010            1010          0111 */
/* 0110    1100            1100          0111 */
/* 0111    1110            1110          0111 */
/* 1000    0000            1111          1000 */
/* 1001    0010            1111          1000 */
/* 1010    0100            1111          1000 */
/* 1011    0110            1111          1000 */
/* 1100    1000            1111          1000 */
/* 1101    1010            1111          1010 */
/* 1110    1100            1111          1100 */
/* 1111    1110            1111          1110 */
```

```
unsigned int USatTimesTwo(unsigned int u) {
    if (u <= 0x7fffffff) {
    return 2*u;
    } else {
    return 0xffffffff;
    }
}
```

```
int TSatTimesTwo(int x) {
    if (x > 0x3fffffff) {
    return 0x7fffffff;
    } else if (x < -0x40000000 /* 0xc0000000 would be unsigned */) {
    return 0x80000000;
    } else {
    return 2 * x;
    }
}
```

Problem 3

The output will be:

```
starting x = 100
starting y = 23
```

```
a) 0x800
b) 0x804
c) 23
d) 0x802
e) -168
f) -168
g) 100
h) -168
```

At first x and y contain 100 and 23, as they were initialized. The addresses of `x`, `px`, `y`, and `py` are `0x800`, `0x802`, `0x804`, and `0x806` respectively, since each is 2 bytes long on this platform and they are allocated sequentially.

(a). Because `px` is initialized as `&x`, its initial value is the address of `x`, namely `0x800`.

(b). Because `py` is initialized as `&y`, its initial value is the address of `y`, namely `0x804`.

(c). The statement `px = py` makes `px` a pointer to the same location that `py` had been pointing to. Specifically `px` now points at `y`. `*px` is the value that `px` points to, namely the value of `y`, or 23.

(d). This statement prints the address of `px`. This never changes, it is still `0x802`.

(e). The statement `*px = -232 + *px` updates the value that `px` points to (i.e. y), to have the value -232 + 23 = -209. However the following statement `y = 41 + *py` changes `y` again, this time to -209 + 41 = -168. `px` is still pointing at `y`, so `*px` is still the value of `y`, -168.

(f). The pointer `py` is still pointing at `y`, so the value of `*py` is the value of `y`, -168.

(g). The variable `x` has not been changed, so it still has the value 100.

(h). From the earlier assignment, `y` has the value -168.

Problem 4

A. In binary, TMax is always single 0 bit followed by 1 bits, and TMin is always single 1 bit followed by 0 bits. So in 8-bit two's complement, TMax is 01111111, and Tmin is 10000000.

B. TMax is $2^7 - 1 = 128 - 1 = 127$. TMin is $-2^7 = -128$.

C.

```
Carries:     11 111
            10110001   -128+32+16+1    =  -79
          + 00110111    32+16+4+2+1    = + 55
          ----------                   -----
            11101000   -128+64+32+8 =     -24
```

No overflow. There is never overflow when adding a negative and a positive number.

D.

```
Carries:         11
            01010001  64+16+1         =      81
          + 00101011  32+8+2+1        = +    43
          ----------                      -----
            01111100  64+32+16+8+4     =    124
```

There is no overflow, hence the result is correct.

E.

```
Carries:   111
             11100000   -128+64+32        =  -32
           + 11111001   -128+64+32+16+8+1 =   -7
           ----------                       -----
             11011001                     =  -39
```

The sum of two negative numbers, has resulted in a negative number (i.e. the sign bit of the result is 1). Hence there is no overflow, and the result is correct.

F.

```
Carries:
             10000000   -128         =  -128
           + 10000000   -128         =  -128
           ----------                   -----
             00000000   0            =   0
```

This is an overflow case. The sum of two negative numbers has resulted in a non-negative number i.e. (sign bit is 0).

G. To negate TMax, we complement it and add 1.

```
TMax      = 01111111 = 127
~TMax     = 10000000 = -128
~TMax+1   = 10000001 = -127

Carries:    1111111
            00000000
          + 10000001
          ----------
            10000001 = -127
```

The negation of 127 is -127.

H. The simplest way to divide by 2 (when dividing an exact multiple of 2), is to shift to the right by one position.

```
( TMin ) >> 1
```

which evaluates to -2^{30}.

Problem 5

A. Rather than going via decimal, the easiest way to convert from hexadecimal to octal is to go via binary: you just have to regroup the bits.

```
0xAB32 = 1010  1011  0011  0010
         1 010 101 100 110 010 (regroup by 3s)
         1   2   5   4   6  2
= 0125462 (octal)
```

But to check, here's the conversion from hex to decimal:

```
0xAB32 = 10*16**3 + 11*16**2 + 3*16 + 2
       =    40960 +     2816 +   48 + 2
       = 43826
```

To convert this to octal, you compute octal digits by repeatedly dividing by 8: the remainders are the digits right to left.

```
43826 = 8*5478+2                                  43826/8 = 5478.25 (.25 = 2/8)
      = 8*(8*684 + 6) + 2                          5478/8 =  684.75 (.75 = 6/8)
      = 8*(8*(8*85 + 4) + 6) + 2                    684/8 =   85.5 (.5 = 4/8)
      = 8*(8*(8*(8*10 + 5) + 4) + 6) + 2             85/8 =   10.625 (.625 = 5/8)
      = 8*(8*(8*(8*(1*8 + 2) + 5) + 4) + 6) + 2  10/8 =    1.25  (.25  = 2/8)
= 0125462 (octal)
```

B. Same approach as in A.

```
0xC23E = 0110  0010  0011  1110
          0 110 001 000 111 110 (regroup by 3s)
          0  2    1   0   7  6
= 021076 (octal)
```

C. Same approach as A. , B.

```
0xB393 = 1011 0011 1001 0011
       =  1 011 001 110 010 011 (regroup by 3s)
       =  1  3   1   6   2   3
= 131623 (octal)
```

D. Same approach as A. , B. , C.

```
0x243A = 0010 0100 0011 1010
       =  0 010 010 000 111 010 (regroup by 3s)
       =  0  2   2   0   7  2
= 022072 (octal)
```

E. Convert to decimal as shown below.

```
B321_16 = 11*16**3 + 3*16**2 + 2*16**1 + 1
        = 45056 + 768 + 32+ 1
        = 45857
```

F. Convert to decimal as shown below.

```
87788_9 = 8*9**4 + 7*9**3 + 7*9**2 + 8*9 + 8
        =  52488 +   5103 +   567 +  72 + 8
        = 58238
```

G. Convert to hex, and then to decimal.

```
 = 1001 1001 1000 1001 (group by 4)

 = 9    9    8    9

 = 0x9989

 = 9*16**3 + 9*16**2 + 8*16 + 9

 = 36864   + 2304 + 128 + 9

 = 39305
```

H. Convert to hex and then to decimal.For signed values, first take the 2s complement of the number if it is negative, and then convert to hex, and finally decimal, remembering to also assign a negative sign.

Unsigned

```
= 10 1010 1100 (group by 4)

= 2   A    C

= 0x2AC

= 512 + 160 + 12

= 684

Answer = 684
```

Signed

```
   0101010011
  +         1
   ----------
   0101010100

= 01 0101 0100 (group by 4)

= 1   5    4

= 1*16**2 + 5*16 + 4

= 256 + 80 + 4

= 340

Answer = -340.
```

I. Same approach as H.

Unsigned

```
= 10 1010 0010 (group by 4)

= 2   A     2

= 2*16**2 + 10*16 + 2

= 512 + 160 + 2

= 674

Answer = 674
```

Signed

```
   0101011101
  +         1
   ----------
   0101011110
```

```
= 01 0101 1110 (group by 4)

= 1   5    14

= 1*16**2 + 5*16 + 14

= 256 + 80 + 14

= 350

Answer = -350.
```

J. Same approach as H. and I.

Unsigned

```
= 11 0001 1100 (group by 4)

= 3   1    C

= 3*16**2 + 1*16 + 12

= 768 + 16 + 12

= 796

Answer = 796
```

Signed

```
   0011100011
  +         1
   ----------
   0011100100

= 00 1110 0100 (group by 4)

=  0  E   4

= 0*16**2 + 14*16 + 4

= 228

Answer =  -228
```